Opinnäytetyö (AMK)

Tietojenkäsittely

2017

Alexander Elo

PROSEDURAALISEN HIIHTOMAAILMAN LUONTI MOBIILIPELILLE UNITY -PELIMOOTTORILLA

– Case Biathlon x2 -peli

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely

2017 | 33

Alexander Elo

PROSEDURAALISEN HIIHTOMAAILMAN LUONTI MOBIILIPELILLE UNITY -PELIMOOTTORILLA

- Case Biathlon x2 -peli

Opinnäytetyön tarkoituksena oli luoda olemassa olevalle mobiilipelille käyttäjäpalautteen perusteella hiihtomaailma. Hiihtomaailman luomisen oli tarkoitus tapahtua proseduraalisesti, jotta pelaajalle voidaan luoda aina uusi, uniikki hiihtomaailma hiihdon alkaessa ja jottei pelin jakeluversion koko kasva paljoa.

Työn teoriaosuudessa tarkastellaan lähtökohtaa, olemassa olevaa tuotetta sekä työkaluja, jolla se on tehty, proseduraalisen generoinnin taustoja ja algoritmeja sekä pelejä, jotka hyödyntävät näitä algoritmeja jollain tavalla. Käytännön osuudessa käydään läpi yksinkertaisen hiihtomaailman toteutusta sekä ratkotaan suunnitteluvaiheen aikana ilmenneitä ajatuksia suorituskyvyn parantamisesta.

Opinnäytetyön lopputuloksena syntyi hiihtomaailman proseduraaliseen generointiin kykenevä järjestelmä, joka täytti määritetyt vaatimukset. Järjestelmää jatkokehitettiin, liitettiin Biathlon x1 -tuotteeseen ja julkaistiin lopulta uutena tuotteena sovelluskauppaan.

ASIASANAT:

Ohjelmistokehitys, algoritmit, peligrafiikka

BACHELOR´S THESIS| ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Batchelor of Business Administration in Information Technology

2017 | 33

Alexander Elo

GENERATING A PROCEDURAL SKIING WORLD FOR A MOBILE GAME IN UNITY GAME ENGINE

- Case Biathlon x2 game

The bachelor thesis's purpose was to create a skiing world for an existing mobile game. Skiing world generation was meant to be procedural so that there would be always a new, unique skiing world upon skiing scene as well as that distribution versions size wouldn't grow much.

On theory section, we inspect the starting point, existing product together with the tools it is made of, procedural generation backgrounds and algorithms as well as game which make a use of procedural generation on some level. Practical section goes through simple skiing world implementation and solves thoughts about performance enhancement that occurred during the planning phase.

The end result was a system capable to generate procedural skiing world which fulfilled the specified requirements. The system was further developed, merged with Biathlon x1 product as was released to the app store as a new product Biathlon x2.

KEYWORDS:

Software development, algorithms, game graphics

SISÄLTÖ

KUVAT

TAULUKOT

1 JOHDANTO

Proseduraalisella generoinnilla voidaan teoriassa saavuttaa loputtomasti eri versioita sisällöstä sekä tuottaa satunnaisuutta, jotta pelattavuus olisi vähemmän ennakoitavissa. Vaikka ensimmäiset pelit, jotka hyödynsivät proseduraalisen sisällön generointia, rakennettiinkin jo 1970 luvulla (Don Worth 12.8.2007), on proseduraalista sisällön generointia käytetty enemmän vasta lähivuosina tuomalla esim. tehtävien sekä esineiden satunnaisuutta. Suosio alkoi kasvaa enemmän 2000 luvun lopulla, jolloin julkaistiin peli, joka hyödynsi proseduraalista generointia koko pelimaailman luomiseen. Nykyään yhä useampi peli hyödyntää proseduraalista generointia entistä enemmän.

Vaikka proseduraalinen generointi ei ole uusi käsite itsessään, lähteiden löytäminen oli haastavaa, sillä aiheesta on kiinnostuttu vasta lähivuosina enemmän. Etenkin kirjallisuutta oli todella niukasti. Jouduin turvautuvaan lähinnä teknisiin asiakirjoihin sekä algoritmien julkaisuihin, lähdekoodiin ja niiden kommentteihin.

Tämän opinnäytetyön tarkoituksena on käydä läpi proseduraalisen hiihtomaailman generoinnin eri toteutusvaihtoehtoja sekä toteuttaa mobiililaitteille vaatimusten mukainen prototyyppi, joka jatkokehitetään ja sovitetaan olemassa olevaan tuotteeseen myöhemmin. Opinnäytetyön tavoitteena oli myös saada lopputulos, joka täyttää toimeksiantajan vaatimukset ja odotukset sekä saada mobiilipelille ennalta-arvaamatonta, pitkäikäistä sisältöä.

Aluksi käydään läpi lähtökohdat, vaatimukset sekä aiempi tuote, johon myöhemmin kyseinen hiihtomaailman generointijärjestelmä liitetään, sillä aiempi tuote määrittää käytettävät työkalut. Samalla pohditaan mahdollisia ongelmia mm. suorituskyvyssä sekä vastauksia niihin teoriatasolla. Sen jälkeen tarkastellaan hieman proseduraalisen generoinnin historiaa ja erilaisia käyttötapoja. Teoriaosuuden viimeisessä vaiheessa tarkastellaan työkaluja ja luokkia sekä kuinka kolmiulotteinen geometria määritetään niillä. Käytännön osuudessa toteutetaan yksinkertainen hiihtomaailman generointijärjestelmä sekä ratkotaan teoriaosuudessa ilmenneitä ongelmia.

.

2 BIATHLON X2 -PELI

Opinnäytetyön toimeksiantaja on RightSpot Oy –niminen yritys Salosta. RightSpot Oy kehittää langattomia ratkaisuja älypuhelimille ja tableteille sekä toteuttaa yritysten tarpeisiin soveltuvia ratkaisuja.

Yrityksen tuote Biathlon X1 on Unity –pelimoottorilla tehty Android- sekä iOS-peli, joka kuvaa ampumahiihtokisojen ampumaosuutta pelattavassa muodossa. Pelissä pelaaja saapuu ampumapaikalle ja tarkoituksena on ampua 50 metrin päässä olevan taulun viisi lätkää alas viidellä patruunalla. Pelissä on viisi tasoa. Jokainen taso sisältää kolme rataa. Mitä korkeammalla pelaaja pelaa, sitä enemmän tähtäin heiluu. Vaikeusaste on kuvattu pelaajan sykkeellä, joka kasvaa tasoon ja rataan nähden.

2.1 Unity-pelimoottori

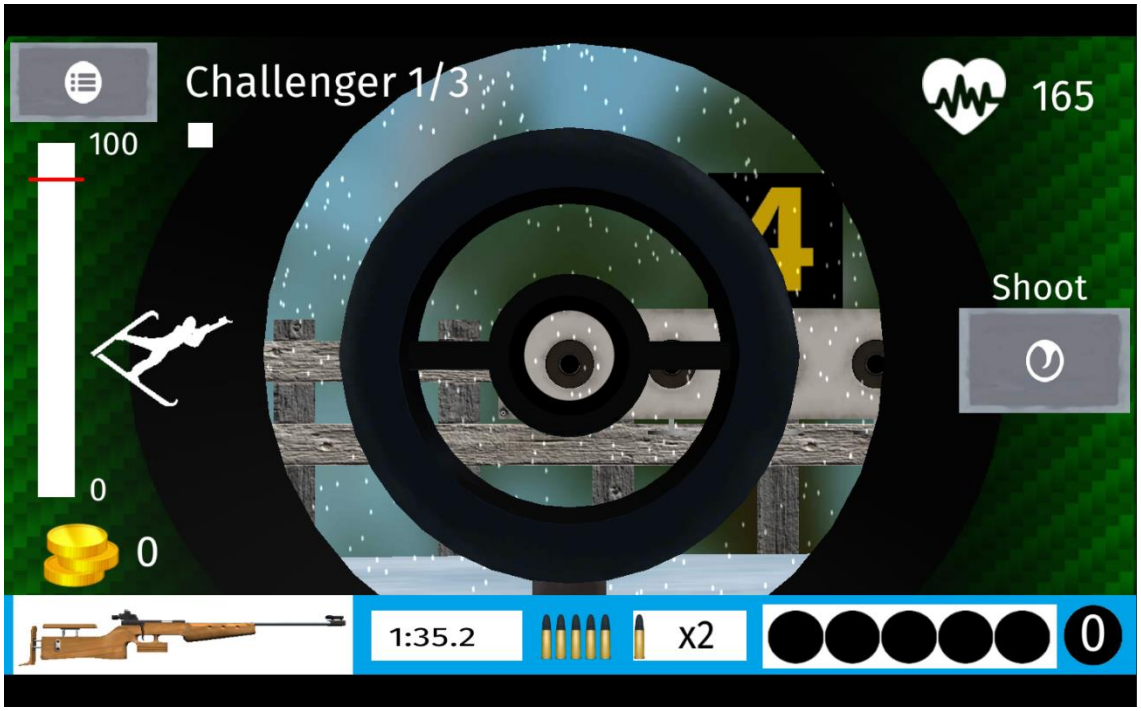
Unity on Unity Technologiesin kehittämä alustariippumaton pelimoottori, jolla voidaan kehittää mm. pelejä PC:lle, konsoleille sekä internetsivuille. Unityllä voidaan tehdä niin kolmiulotteisia kuin kaksiulotteisia pelejä, virtuaalitodellisuuteen perustuvia pelejä sekä lisätä erilaisiin näkymiin, kuten läpikatseltavien näyttöjen kautta, tietokonegrafiikalla tuotettuja elementtejä.

Unityn ominaisuudet ovat kattavat. Se sisältää mm. fysiikkamoottorin (NVIDIA PhysX 3.3), reaaliaikaisen äänien miksauksen, hiukkastehosteet sekä sillä voidaan tehdä animaatioita editorin sisällä. Mahdolliset puutteelliset toiminnot voi tehdä itse tai ostaa kolmansilta osapuolilta Unityn Asset Store:sta. (Unity 2017a)

Unitystä on saatavilla erilaisia versioita, joista alin on ”Personal” luokka. Personal versio on ilmainen aina 100 000 dollarin vuotuiseen tuottoon saakka. Personal luokan käännettyihin peleihin tulee ”Made with Unity” aloitusruutu, jota ei saa pois. Maksullisissa versioissa kuten Plus, Pro sekä Enterprisessä aloitusruutu on täysin muokattavissa. Halvimmassa Plus versiossa on nostettu vuotuinen tuloraja 200 000 dollariin, kun muissa versioissa rajaa ei ole ollenkaan. (Unity, 2017b)

2.2 Biathlon x1 -pelin kulku

Ammunnassa peli simuloi tähtäimen heilunnan ja pelaajan on ajoitettava ampuminen oikein. Jokaisen lätkän kohdalla on ajastin, joka mittaa tähtäysajan. Mitä nopeammin ampuu osuman, sitä enemmän saa kolikoita. Ohilaukauksen sattuessa pelaaja menettää osan potentiaalisesta kolikkosaatavasta sekä voi ampua uudelleen kolmen varapatruunan turvin.



Kuva 1. Kuvankaappaus Biathlon x1 pelistä.

Mikäli patruunat loppuvat kesken, pelaaja voi joko ostaa lisäpatruunoita kolikoilla, tai aloittaa radan uudelleen. Jokaisella radalla on kaksi tai useampi ampuma-asentoa. Makuuasennossa ammuttaessa osuma-alueen koko on 4,5 cm ja pystyasennosta ammuttaessa 11.5 cm. Pystyammunnassa tähtäimen heilunta on suurempi. Jokaisen radan loputtua pelaaja sijoitetaan tuloslistaan, jossa näytetään pelaajan nimi, aika, sekä ohi-laukaukset. Pelaaja voi myös ostaa oikealla rahalla pelin sisäisiä kolikoita, joilla voi ostaa mm. patruunoita tai erilaisia aseita, kuten haulikon tai konekiväärin.

2.3 Toimeksianto

Pelin käyttäjäpalautteen perusteella Biathlon x1 tuotteeseen kehitetään hiihto-osio. Hiihto-osion myötä peli muuttuu niin paljon, että siitä tehdään uusi tuote Biathlon x2.

Hiihto-osio tulee olemaan ns. endless runner -tyyppinen, jossa pelaajan tavoitteena on selviytyä mahdollisimman pitkään väistämällä esteitä ja keräämällä kolikoita. Pelaaja ei pysty kontrolloimaan hahmon hiihtonopeutta. Radalla tulee olemaan kolme hiihtolatua. Pelaajan tulee voida pyyhkäistä mobiililaitteen näyttöä horisontaalisti, jolloin hahmon hiihtolatu vaihtuu toiseen. Endless Runner -tyyppisistä peleistä poiketen pelin vaikeus ei tule kasvamaan ajan myötä eikä hiihto-osuus ole loputon. Lisäksi esteisiin osuminen ei lopeta pelisessiota, vaan käyttäjä saa sanktioita hiihtonopeuden hidastumisella sekä kolikoiden menettämisellä.

Radan alkuun ja loppuun sekä ammuntojen väliin sijoitetaan hiihto-osuus. Hiihto-osuuden pituus sekä ammuntojen määrä tulee määrittymään tason mukaan. Mitä vaikeammalla tasolla pelaaja tulee pelaamaan, sen enemmän ammuntakertoja sekä pidempi hiihtokokonaisuus.

Pelin vaikeuttamiseksi jokaiselle hiihtoladulle sijoitetaan objekteja eri kohtiin. Pelaajan tulee väistää tulevat esteet sanktion välttämiseksi, sekä kerättävä kolikoita kassan kerryttämiseksi. Esteistä tulee olemaan useita erilaisia variaatioita, mutta ne kuuluvat aina vähintään yhteen elementtiin:

- Hyppy: Este on vähintään kolmen ladun levyinen. Pelaajan on hypättävä esteen yli pyyhkäisemällä ylöspäin.
- Väistö: Este on maksimissaan kahden ladun levyinen. Pelaajan on väistettävä estettä vaihtamalla rataa pyyhkäisemällä vasemmalle tai oikealle.
- Kumarrus: Este on korkealla. Pelaajan on pyyhkäistävä alaspäin, jotta hahmo kumartaa ja pääsee esteen ali.

2.4 Pelikenttä

Kolmiulotteinen pelikenttä tullaan generoimaan proseduraalisesti. Näin ei jokaiselle hiihtoradalle joudu tekemään käsin omaa rataa, vaan se voidaan generoida erilaisilla säännöillä. Sääntöjä hienosäätämällä voidaan rakentaa joka kerralla erilainen pelikenttä. Pelikenttä tulee sisältämään kaksi pääkomponenttia: ulkomaailma sekä hiihtoalue.

Ulkomaailma generoidaan kohina-algoritmia hyödyntäen pehmeämuotoisiksi mäiksi, jotka tulevat sisältämään erilaisia koristeita, kuten taloja ja puita. Ulkomaailman ainut funktio on olla ns. somiste, joten esimerkiksi fysiikka ja törmäystarkastelu (collision detection) voidaan kytkeä pois sekä ulkomaailmalta että siinä sijaitsevilta staattisilta objekteilta.

Hiihtoalue rakennetaan ulkomaailman keskelle. Ulkomaailma leikataan kahtia pituussuunnassa. Ulkomaailman puolikkaita siirretään niin, että leikkauskohdalle jää hiihtoradan levyinen tila. Tämä tila täytetään hiihtoradalla, jonka korkeus riippuu ulkomaailman leikkauskohdan korkeudesta, jolloin erilliseltä hiihtoradan generoinnilta säästytään.

2.5 Suorituskyky

Koska peli on tarkoitettu mobiililaitteille, on suorituskyky rajoitettu. Vaikka nykyään mobiililaitteilla onkin useita säikeitä sekä useampi gigatavu keskusmuistia, on huomioitava myös ns. ”low-end”-laitteet sekä vanhemmat mobiilimallit saavuttaakseen maksimaalisen laitelevikin.

Ulkomaailman generointi Unity:n omalla kohina-algoritmilla on prosessori-intensiivistä laskelmointia. Koska useammilla mobiililaitteilla on säikeitä enemmän kuin yksi, voidaan laskelmointia nopeuttaa hyödyntämällä moniajoa, jolloin prosessorin eri ytimet laskevat eri osaa ulkomaailmasta. Näin saadaan mahdolliset hiihtoskenaarion latausajat pienemmäksi.

Mobiililaitteiden grafiikkapiiri ei myöskään pysty piirtämään näytölle loputtomasti tarkkaa kuvaa, vaan polygonien kasvaessa myös piirtoaika kasvaa. Liian hidas piirtoaika saa pelin näyttävän pätkivältä, joten koko pelimaailman luomisen jälkeen se tullaan jaka-

maan yhtä suuriin paloihin, ns. chunkeihin, joita aktivoidaan tarpeen mukaan, kun pelaaja liikkuu eteenpäin. Näin myös vältytään instantioimisen viiveeltä, jolloin peli pysähtyy muutamiksi millisekunneiksi.

3 PROSEDURAALINEN GENEROINTI

Proseduraalinen generointi on yksinkertaisuudessaan algoritmi, joka generoi satunnaisesti pelisisältöä. Koska pelisisältö on satunnaista, voidaan joka pelikerralla huomata uutta sisältöä. Esimerkiksi koko tai osittaisen pelimaailman proseduraalisella generoinnilla säästetään kehitysajassa sekä valmiin pelin koossa, sillä algoritmit voivat generoida uutta sisältöä sitä tarvittaessa. Lisäksi pelaajalla on käytännössä loputon määrä eri variaatioita pelikentästä.

Täysi satunnaisuus ei ole hyvä. Jos peli generoi pelin komponentteja täysin satunnaisesti, ei niissä komponenteissa olisi mitään järkeä ja peli olisi pelkkää kaaosta. Siksi proseduraalinen generointi hallitsee satunnaisuutta ja saa satunnaisuuden noudattamaan joitakin perussääntöjä.

3.1 Satunnaislukugeneraattorit

Pelikehityksessä on käytetty proseduraalista generointia jo kymmeniä vuosia. Yksi ensimmäisistä peleistä, jotka on tehty proseduraalista generointia hyödyntäen, on Beneath Apple Manor. 1970-luvun lopussa tehty peli loi pelikentät algoritmilla, joka tulosti ASCII-merkistön merkkejä luoden huoneita ja luolastoja, joita pelaaja voi tutkia. (Don Worth 12.8.2007) Tämä antoi pelille pitkän eliniän, sillä erilaisia huoneita ja luolastoja oli käytännössä loputon määrä, luoden jokaiselle pelikerralle uutta sisältöä.

Kuva 2. Kuvakaappaus Beneathe Apple Manor -pelin ensimmäisestä julkisesta versiosta.

Manuaalisen, staattisen datalohkon sijasta sisältö voidaan siis luoda erilaisilla algoritmeilla. Algoritmit sisältävät erilaisia sääntöjä, joita voidaan muokata tarvittaessa erilaisilla muuttujilla sekä satunnaisgeneraattorilla saaden hieman erilaisia tuloksia joka kerralla. Esimerkiksi ”Beneath Apple Manor” -pelin huonegeneraattorille voitaisiin syöttää yllä olevan kuvan mukaan seuraavat säännöt saaden samantapainen tulos:

- Tee 4-8 kappaletta huoneita
- Yhden huoneen leveys ja pituus ovat 1-5 yksikköä
- Huoneiden on oltava kiinni toisissaan

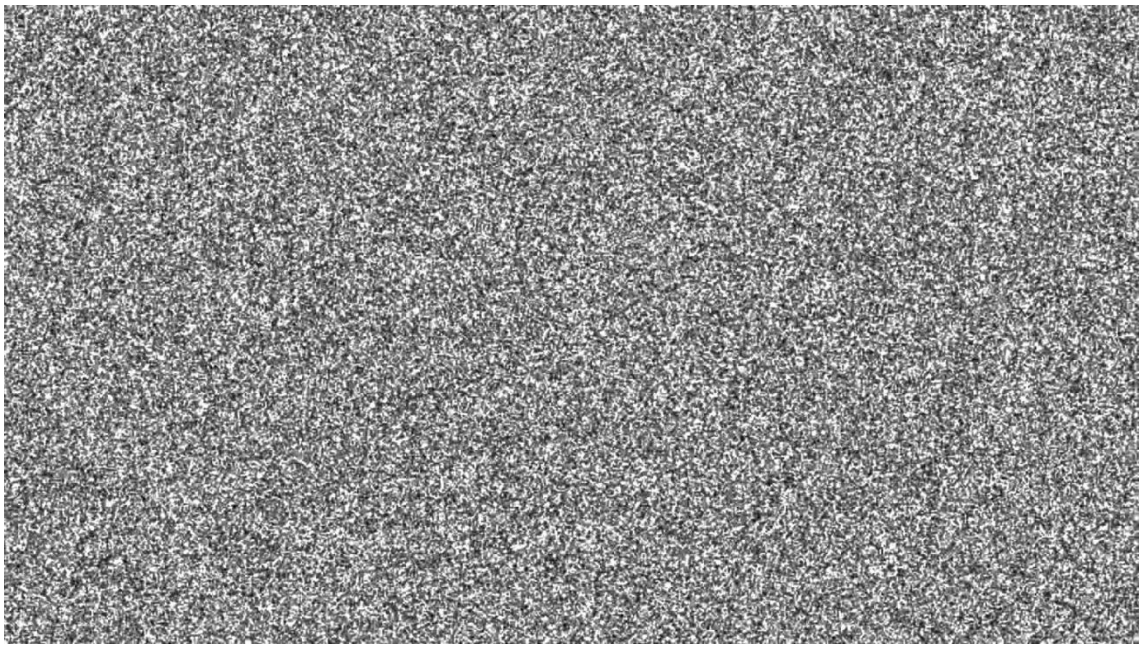
Tämä huonegeneraattori ei kuitenkaan luo jokaisella kerralla samaa pelikenttää, sillä esimerkiksi huoneet voivat olla eri järjestyksessä ja erikokoisia.

Yleisimmät satunnaislukugeneraattorit eivät ole täysin satunnaisia. Nämä pseudosatunnaiset generaattorit perustuvat melkein aina matemaattisiin kaavoihin sekä ennalta määritettyihin lukuihin. (Goldreich 2006, 287-291.) Kun nämä ennalta määritetyt luvut ja säännöt tietää, voidaan aina luoda sama sekvenssi. Joillekin algoritmeille ja satun-naislukugeneraattoreille voidaankin yleensä määrittää siemenarvot, joiden mukaan nämä alkuarvot arvotaan. Näin ollen samaa siemenlukua käyttäen voidaan aina luoda samat

sekvenssit uudelleen ja uudelleen. Tätä voidaan hyödyntää esimerkiksi monipeleissä, jolloin varmistetaan, että pelaajille luodaan tarvittaessa samanlainen sisältö.

3.2 Kohina-algoritmit

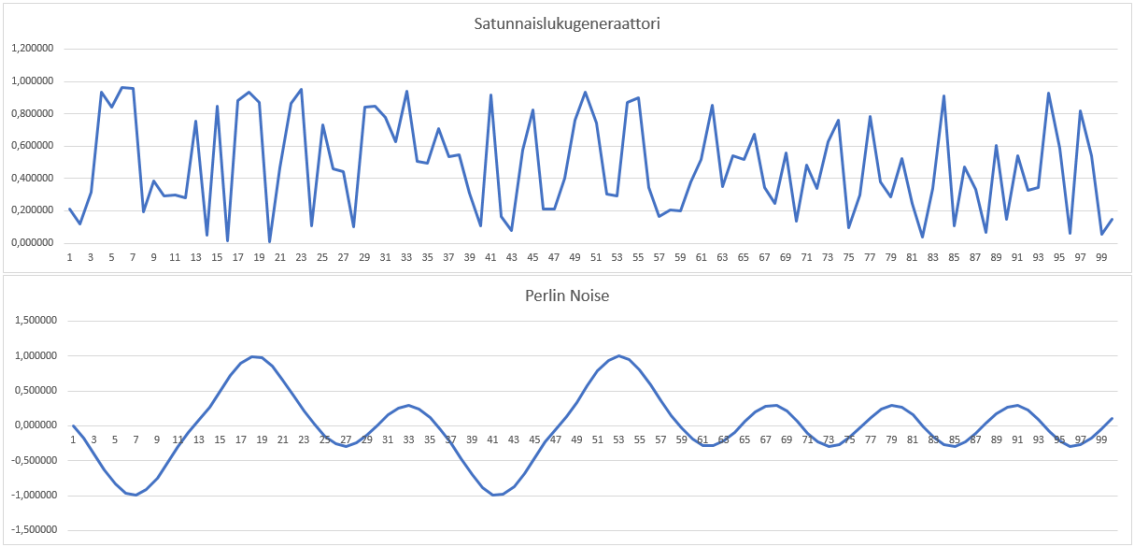
Kohina on sarja satunnaisia lukuja, jotka järjestetään yleensä linjaksi tai moniulotteiseksi taulukoksi (Shaker ym. 2016, 58). Esimerkiksi analogisen tv-verkon aikoihin katsoessaan taajuutta, jota ei lähetetty, nähtiin pelkkää kohinaa – kaksiulotteisen taulukon mustista ja valkoisista pisteistä.



Kuva 3. Kuvakaappaus kohinasta, joka koostuu mustista ja valkoisista pisteistä.

Pelkästään satunnaislukugeneraattorin antamat luvut eivät yleensä sovellu sellaisenaan esimerkiksi pelikentän generoimiseen. Satunnaislukugeneraattori antaa joukon satunnaisia lukuja, joiden välillä ei ole mitään yhteyttä. Pelin Noise-kohina-algoritmi antaa myös satunnaisia lukuja, mutta pisteillä on yhteys naapurin arvoihin. Mitä lähempänä pisteet ovat toisiaan, sen samanarvoisempia ne ovat (Perlin 2001).

Taulukko 1. Satunnaislukugeneraattorin ja Perlin Noise -algoritmin arvojen vertailu.

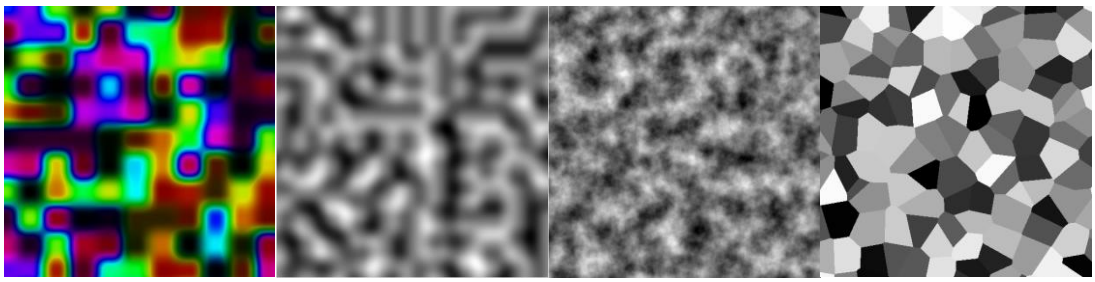


Perlin Noise -kohina-algoritmia kehitti Ken Perlin vuonna 1985. Tätä ensimmäistä tietokoneella generoitua kohina-algoritmia kehitettiinkin alun perin luonnollisuuden saavuttamiseksi tietokonegrafiikoissa, esimerkiksi tekstuureissa tai animaatioissa. (Perlin, 2001) Yksiulotteista Perlin Noise -kohinaa voidaan hyödyntää esimerkiksi viivan poikkeamina luoden käsinkirjoitetun tunnelman. Kaksiulotteista Perlin Noise -kohinaa voidaan hyödyntää esimerkiksi kivi- tai tulitekstuurin luomisessa. Hieman kolmiulotteista Perlin Noise -algoritmia muokaten voidaan luoda kokonainen maailma luolastoilla ja onkaloilla. Tästä hyvänä esimerkkinä ruotsalaisen yhtiön Mojangin kehittämä suosittu peli Minecraft, joka generoi maailman kolmiulotteista Perlin Noisea hyödyntäen. Microsoft osti Minecraftin 2,5 miljardilla dollarilla vuonna 2015. (Microsoft, 2014) Peliä on ostettu yli 26 miljoonaa kappaletta (Minecraft, 2017).

Kuva 4. Kuvankaappaus Minecraft pelistä.

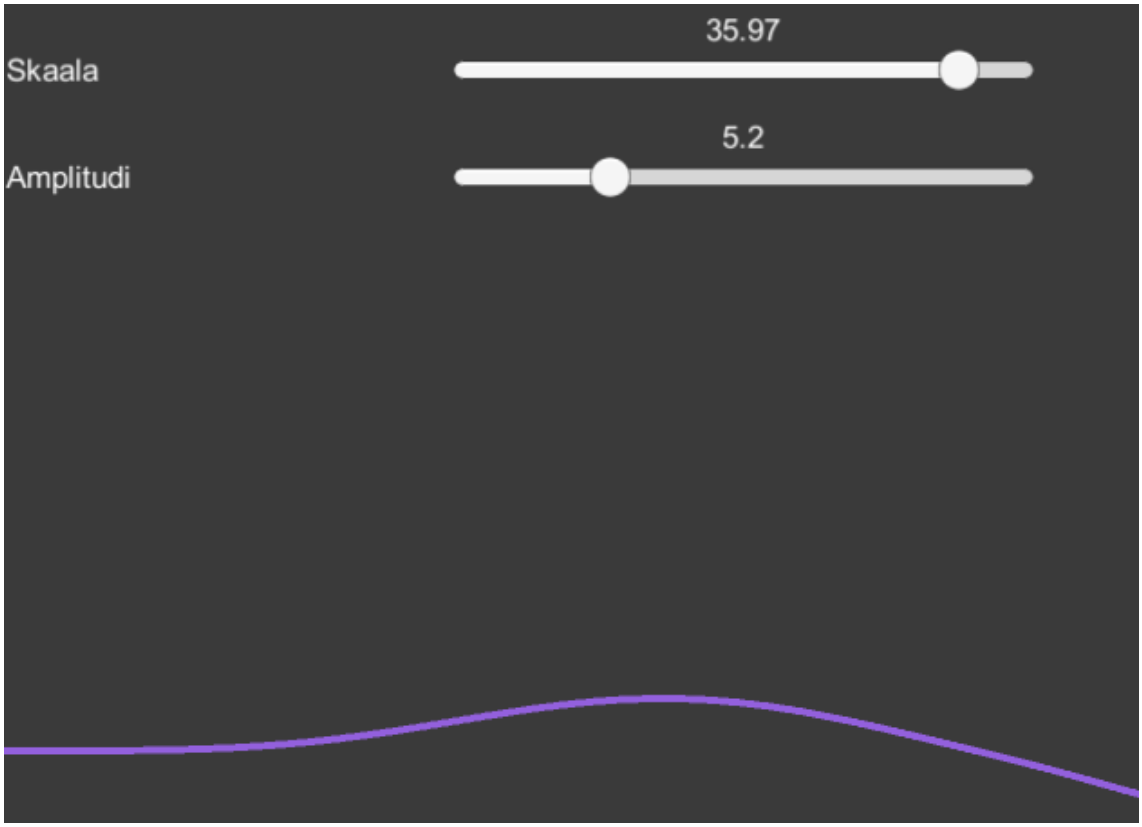
Ehkä yksi Minecraftin ison suosion syistä on se, ettei siitä lopu sisältö. Jos pelin pelimaailmaan kyllästyy, voi Minecraftissa aloittaa uuden pelin, jolloin tietokone rakentaa kokonaisen uuden maailman. Nämä maailmat ovat valtavia ja sisältävät kauniita alueita, jotka ovat maantieteellisesti erillään. Generoidussa maailmassa erilaiset biomit, kuten tropiikki, tundra, metsä tai vuoristoinen maasto, luovat erilaisen tunnelman eri alueilla. Yhden Minecraft maailman pinta-ala on noin 3 600 000 000 km^2 kun maapallon on 510 100 000 km^2.

Vaikka suosituimpia algoritmeja onkin kourallinen, niiden tulostamien arvojen käyttötapoja on monia. Ehkä helpoin ja yksinkertaisin tapa on käyttää Perlin Noise -algoritmia ja syöttää sen parametreihin X ja Z -koordinaatit ja käyttää saatavaa arvoa Y:nä, eli kyseisen pisteen korkeutena. Käytettäessä X:n ja Z:n mahdollisina arvoina kokonaislukuja 0-32 ($i \in Z: 0 \leq i \leq 32$) saadaan yli 1 000 m^2 alueen korkeusarvot.



Kuva 5. Erilaisten kohina-algoritmien tuloksia.

Kuva 6:ssa on käytetty Unityn yksiulotteista Perlin Noisea, jolle on syötetty X akselin koordinaatti ja saatua arvoa on käytetty Y akselin arvona eli korkeusarvona. X akselin koordinaatti jaetaan skaalan luvulla sekä saatu korkeusarvo kerrotaan amplitudin luvulla.



Kuva 6. Unityn Perlin Noise algoritmin tuloste korkeusarvoina.

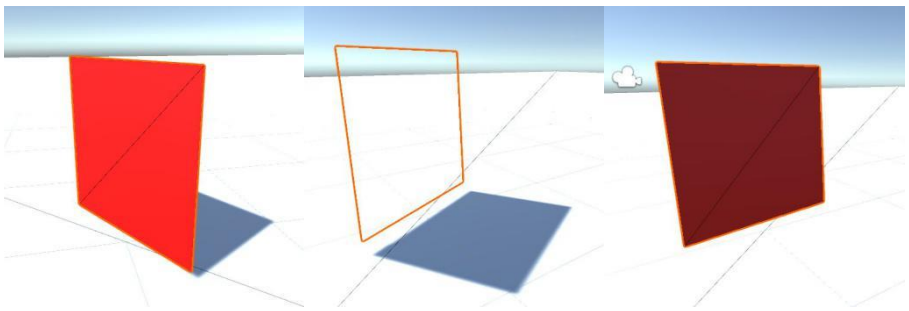
4 UNITYN MESH -LUOKKA

Mesh-luokka mahdollistaa komentosarjojen pääsyn objektien geometriaan. Mesh-luokan avulla voidaan muokata olemassa olevaa kolmiulotteista objektia tai rakentaa se tyhjästä. Mesh luokka pitää sisällään mm. Verteksit, kolmioindeksit, UV koordinaatit, normaalit ja tangentit. Jokaisella verteksillä voi olla normaali, kaksi tekstuurikoordinaattia, värikoodi sekä tangentti. Vaikka periaatteessa kolmion piirtämiseen riittää pelkästään verteksien ja indeksien määrittäminen, on hyvä myös lisätä normaalit sekä UV koordinaatit, jotta objekti reagoi valoon sekä osaa näyttää tekstuurin oikein.

4.1 Verteksit ja kolmiot

Unityn kolmiulotteinen objekti koostuu pääosin erikokoisista kolmioista, jotka ovat järjestetty oikein verteksien kesken kolmiulotteisessa avaruudessa. Kolmion muodostamiseksi tarvitaan taulukko vertekseistä sekä verteksi-indekseistä. Verteksitaulukko määrittää kulmapisteet ja verteksi-indeksitaulukko määrittää missä järjestyksessä kolmioita piirretään verteksitaulukon pisteiden välillä. Koska kolmiolla on kolme verteksiä, on indeksitaulukon koko oltava kolmella jaollinen. Kolmiot voivat jakaa samoja verteksejä keskenään.

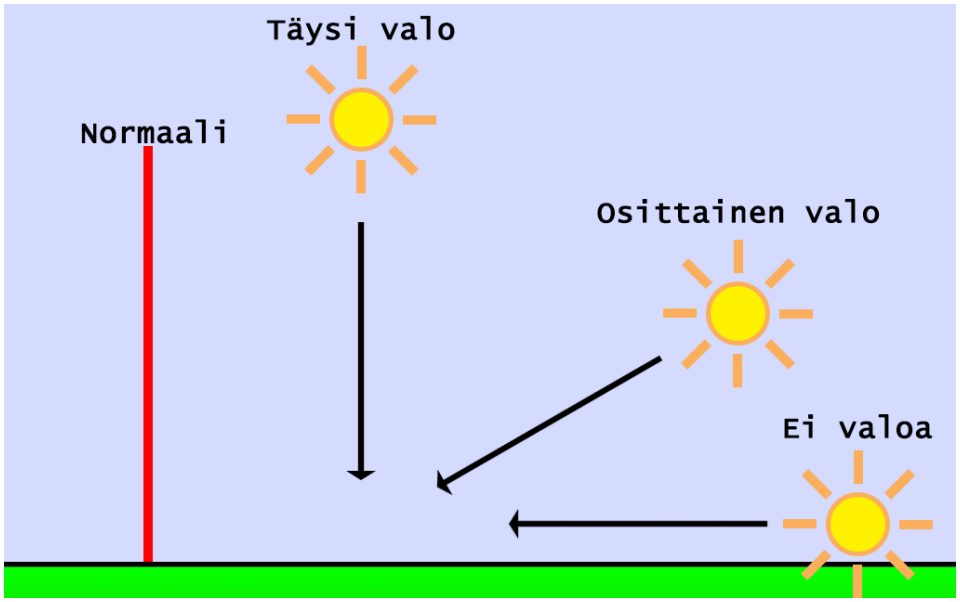
Kuten kuvan 7 kaksi ensimmäistä osioita näyttää, pitää muistaa, että polygoni on kiinteä vain yhdeltä puolelta. Mikäli verteksien indeksit määritetään kellon suuntaan, on polygoni nähtävissä etupuolelta, mutta ei takapuolelta. Sama pätee varjoihin, takapuoli ei ole kiinteä, joten se ei myöskään luo varjoa (kuva 7, viimeinen osio).



Kuva 7. Polygonin käyttäytyminen eri suunnista.

4.2 Normaalit

Jotta kolmion pinta reagoi valoon, määritetään jokaiselle kolmion kulmalle (verteksille) normaali vektori. Normaalin vektori osoittaa vektorista ulospäin. Valon laskemisen aikana verrataan verteksin ja valon vektoreita toisiinsa: mikäli verteksit ovat täydellisesti linjassa saa verteksi täyden kirkkauden valosta. Mikäli linja ei ole täysin suora niin valon määrä lasketaan kahden kyseisen verteksin määrittämästä asteluvusta, jolloin valon määrä on jotakin täyden valon ja täyden varjon välillä. Kolmion kulmapisteiden valomäärät interpoloidaan kolmion pintaan.



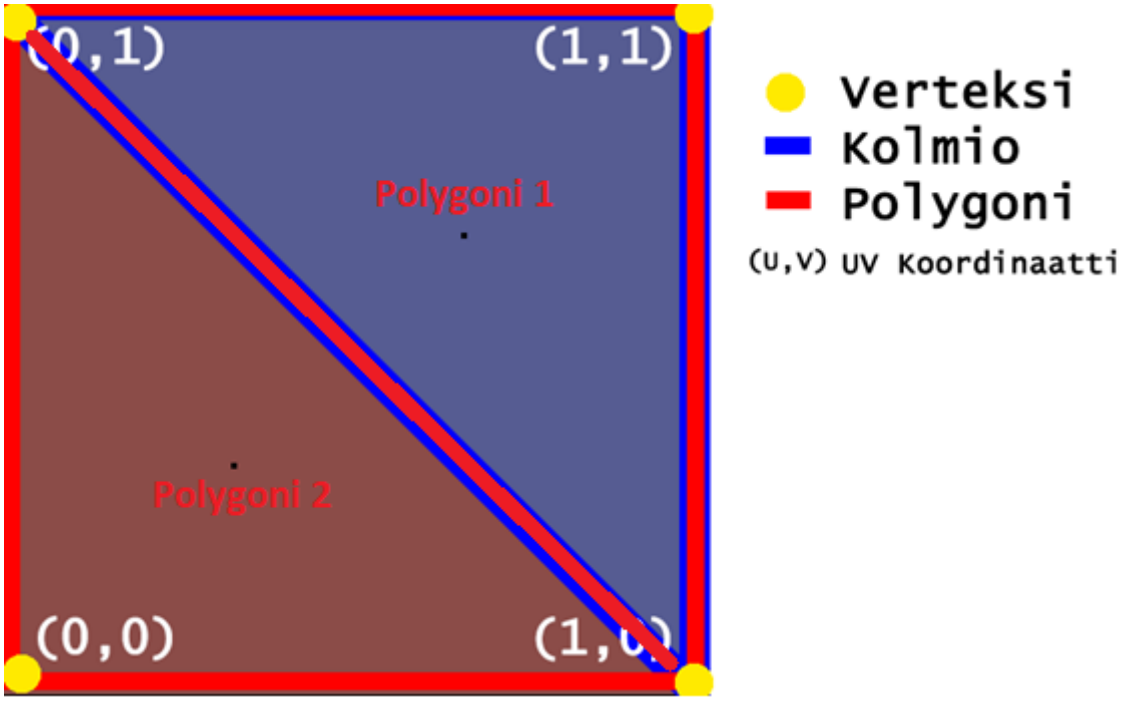
Kuva 8. Valon käyttäytyminen normaalin mukaan.

4.3 UV -koordinaatit

UV koordinaatit määrittävät kuinka kaksiulotteinen tekstuurikuva (esimerkiksi ruoho) sovelletaan kolmiulotteiseen pintaan. Tekstuuri on kuin kuva tulostettu venyvään materiaaliin, joka venytetään pintaan kiinni. Jokaiselle verteksille määritetään UV -koordinaatti.

Tekstuurikuvan koordinaatit ovat kaksiulotteisia ja arvot ovat 0..1, jossa 0 tarkoittaa vasen-/alanurkkaa ja 1 oikea-/ylänurkkaa. Nämä koordinaatit kertovat missä kohtaa polygonin verteksit ovat tekstuurikuvassa. Esimerkiksi neliön muotoisen polygonin UV

koordinaatit ovat (0,0), (0,1), (1,1) ja (1,0), jolloin jokainen polygonin kulma on myös tekstuurin vastaava kulma.



Kuva 9. Verteksin, kolmion, polygonin sekä UV koordinaattien määritys.

5 HIIHTOMAAILMAN TOTEUTUS

Toteutus tehdään neljässä osassa:

- Settings-luokka, jolta voidaan kysellä sekä tallentaa asetuksia, kuten kohinan skaalaa ja amplitudia. Settings-luokka hyödyntää Unityn PlayerPrefs luokkaa, joka mahdollistaa erilaisten tietojen säilyttämistä eri pelisessioiden välillä.
- WorldObjects-luokka, joka pitää yhden kentän osan objektit (maailman geometria, esteet, kolikot...) yhdessä kasassa. Kun aktivoidaan yksi WorldObjects-luokan objektit, aktivoituu kaikki tarvittava yhdestä kentän osasta.
- BiathlonNoise-luokka pitää huolen kohinan laskemisen oikeasta tavasta ja antaa ulostulona aina oikean maaston korkeusarvon. Näin muut luokat, jotka hyödyntävät kohinaa, voivat keskittyä vain omiin toimintoihin.
- Chunk-luokka, joka pitää kentän geometrian, tarvittavat geometrian piirtämisen luokat sekä muut tarvittavat maaston tiedot. Lisäksi Chunk-luokka pitää huolen siitä, että geometrian ja kohinan laskeminen tapahtuu usealla säikeellä.

5.1 Settings-luokka

Settings-luokalla on kolme tehtävää: lukea asetuksia, tallentaa asetuksia sekä palauttaa oletusasetuksia, mikäli tietoja ei ole muutettu ja tallennettu muistiin. Unityn PlayerPrefs luokka sisältää metodit kuten GetFloat, GetString sekä GetInt, joiden avulla haetaan oikea datatyyppi. Vaikka PlayerPrefs-luokka sisältääkin HasKey metodin, jolla voidaan tarkistaa, onko kyseinen asetus tallennettu muistiin, voidaan myös asettaa palautettava oletusarvo Get-metodeille, joka palautetaan, mikäli kyseistä asetusta ei löydy muistista. Näin ollen Settings-luokka voi yksinkertaisuudessaan hyödyntää C# ominaisuuksia, jotka tarjoavat joustavan tavan lukea sekä kirjoittaa yksityisten kenttien arvoja. Ominaisuuksilla on aksessorit get ja set, joilla voidaan määrittää PlayerPrefs-luokan luku ja kirjoitukset.

Kuvassa 10 näkyy osa toteutetusta Settings-luokasta, jossa on määritetty kohinan skaala- ja amplitudiominaisuudet. Haettaessa arvoa, huomataan get:n palauttavan

PlayerPrefs.GetFloat tulostaman arvon, joka haetaan asetuksista - tai palautetaan oletusarvo 35,873 mikäli asetusta ei löydy tallennettuna.

```
1   using UnityEngine;
2
3   public static class Settings
4   {
5       public static float NoiseScale
6       {
7           get { return PlayerPrefs.GetFloat("NoiseScale", 35.873f); }
8           set { PlayerPrefs.SetFloat("NoiseScale", value); }
9       }
10
11      public static float NoiseMultiplier
12      {
13          get { return PlayerPrefs.GetFloat("NoiseMultiplier", 5.231f); }
14          set { PlayerPrefs.SetFloat("NoiseMultiplier", value); }
15      }
16  }
```

Kuva 10. Settings-luokka.

5.2 WorldObjects-luokka

Jotta peli pyörisi paremmin eikä kuluttaisi laitteen akkua laskemalla resursseja, joita ei pelaaja näe, jaetaan kenttä ja sen objektit tasaisiin osiin. WorldObjects-luokka pitää näistä osista kirjaa. Aktivoimalla yhden WorldObjects-luokan objektit voidaan aktivoida koko kentän osan objektit.

Aktivointi tapahtuu pelaajan position mukaan. Peliin asetetaan, kuinka pitkälle pelaajan pitäisi nähdä. Tämän rajan toinen puoli on sumun takana piilossa, jolloin pelaaja ei näe mitä sen takana tapahtuu. Näin ollen voidaan sumun takana olevat objektit piilottaa sekä poistaa käytöstä, jolloin säästetään niin prosessorin kuin näytönohjaimen rasituksesta. Samalla voidaan pelin latauksen aikana generoida koko maailma eikä peli pysähdy vanhemmilla laitteilla pieneksi hetkeksi geometrian generoimisen sekä objektien sijoittamisen ajaksi kesken pelin.

WorldObjects-luokka pitää siis yksinkertaisuutenaan kirjaa yhden pelikentän osan geometriasta sekä sen päälle tulevista objekteista. Koska geometriaa emme tule manipu-

loimaan generoimisen jälkeen, voidaan suoraan referoida kentän geometrian GameObjectiin, jonka aktivointistatusta voidaan helposti muuttaa. WorldObject:n metodeihin tulee ainoastaan mahdollisuus lisätä objekteja lisää kyseiseen kentän osaan sekä vaihtaa osan objektien aktivointistatusta. Lisäksi WorldObjects sisältää yhden ominaisuuden, jolla nähdään, onko kyseisen osan objektit aktiivisia vai ei. Tämä ominaisuus tulisi vaihtua automaattisesti, kun statusta muutetaan eikä se ole muulla tavalla vaihdettavissa.

```
1   using System.Collections.Generic;
2   using UnityEngine;
3
4   public class WorldObjects
5   {
6       public bool Active { get; private set; }
7
8       private readonly GameObject _worldGeometry;
9       private readonly List<GameObject> _objects;
10
11      public WorldObjects(GameObject geometry)
12      {
13          _worldGeometry = geometry;
14          _objects = new List<GameObject>();
15          Active = true;
16      }
17
18      public void AddObject(GameObject newObject)
19      {
20          _objects.Add(newObject);
21      }
22
23      /// <summary>
24      /// Changes whole WorldObjects chunk objects active status
25      /// </summary>
26      /// <param name="active">Should objects be active or not</param>
27      public void SetActive(bool active)
28      {
29          _worldGeometry.SetActive(active);
30
31          foreach (GameObject gameObject in _objects)
32          {
33              gameObject.SetActive(active);
34          }
35
36          Active = active;
37      }
38  }
```

Kuva 11. WorldObjects-luokka.

5.3 BiathlonNoise-luokka

BiathlonNoise -luokan tehtävänä on hoitaa kaikki kohinaan liittyvä. BiathlonNoisen GetHeight-metodille syötetään haluttu X ja Z koordinaatti, jolloin saadaan kyseisen koordinaatin korkeusarvo. Lisäksi, jotta jokainen kenttä ei olisi samanlainen, lisätään vielä satunnaisgeneraattorille siemenarvo sekä ns. Offset arvo, joka siirtää kohinan aloituspistettä muualle kuin (0,0) pisteeseen. Tämä tuo lisää kohinan satunnaisuutta.

Koska työssä käytetään useita säikeitä geometrian laskemiseen ja generoimiseen, meidän pitää tallettaa kohinan skaala ja amplitudi omiin muuttujiin (kuva 12, rivi 24-25). Suurin osa Unityn omista luokista toimii ainoastaan, kun kutsu tulee Unityn pääsäikeestä. Kutsu näihin luokkiin (tässä tapauksessa PlayerPrefs-luokka) toisista säikeistä heittää poikkeuksen.

Kuvan 12 rivillä 10 alkaa siemenarvon asettaminen sekä satunnaislukugeneraattorin alustaminen oikealla siemenarvolla. Siemenarvon asettaminen asettaa myös uudet offset arvot, jotka valitaan satunnaisesti satunnaislukugeneraattoria hyödyntäen. Näin saadaan samalla siemenarvolla aina sama korkeusarvo.

GetHeight -metodille (kuva 12, rivi 43) syötetään pelimaailman X sekä Z koordinaatit ja saadaan kyseisen pisteen korkeusarvo. Ennen X ja Z koordinaatin syöttämistä ne käsitellään seuraavasti:

- Koordinaatin piste jaetaan kohinaskaalalla. Isompi skaala venyttää ja tasoittaa kohinaa. Suurempi skaala kasvattaa kahden vierekkäisten mäkien huipun välillä olevaa matkaa.
- Koordinaatin pisteeseen lisätään offset arvo, joka siirtää kohinan alkupistettä muualle. Näin saadaan hieman enemmän satunnaisuutta.
- Saatu arvo kerrotaan kohinan amplitudilla, jolla voidaan säätää rinteiden jyrkkyyttä.

```
using UnityEngine;

namespace Noise
{
    public static class BiathlonNoise {

        /// <summary>
        /// Sets noise and RNG seed value as well as noise offset and scale/multiplier values
        /// </summary>
        public static int Seed
        {
            get
            {
                return _seed;
            }

            set
            {
                _seed = value;

                Random.InitState(_seed);
                NoiseOffsetX = Random.Range(-100000f, 100000f);
                NoiseOffsetZ = Random.Range(-100000f, 100000f);
                _noiseScale = Settings.NoiseScale;
                _noiseMultiplier = Settings.NoiseMultiplier;
            }
        }

        private static float NoiseOffsetX { get; set; }
        private static float NoiseOffsetZ { get; set; }

        private static int _seed;
        private static float _noiseScale;
        private static float _noiseMultiplier;


        /// <summary>
        /// Get world height on specific point
        /// </summary>
        /// <param name="x">X coordinate</param>
        /// <param name="z">Z coordinate</param>
        /// <returns>Height value of Y</returns>
        public static float GetHeight(float x, float z)
        {
            return Mathf.PerlinNoise(
                        x / _noiseScale + NoiseOffsetX,
                        z / _noiseScale + NoiseOffsetZ)
                    * _noiseMultiplier;
        }
    }
}
```

Kuva 12. BiathlonNoise-luokka.

5.4 Chunk-luokka

5.4.1 Muuttujat ja alustusmetodi

Kuten aiemmin viittasin Unityn Mesh luokkaan, tarvitsemme sen, jotta voimme manipuloida ja luoda tyhjästä 3D geometriaa. Verteksien ja kolmioindeksien lisäksi tarvitsemme normaalit, jotta valo reagoin maaston pintaan oikein, sekä UV koordinaatit, jotta maastolle saadaan lisättyä tekstuurit oikein.

Geometria siis tulee sisältämään verteksit, kolmioindeksit sekä UV koordinaatit. Mesh -luokka vastaanottaa nämä datat taulukkona, mutta taulukoissa on se ikävä puoli, että niiden koko täytyy olla ennalta määritetty. Jotta taulukoiden koon laskemiselta vältytään voidaan käyttää System.Collection.Generics nimiavaruudesta löytyvää List<T> luokkaa. List<T> voidaan pitää dynaamisena taulukkona, jonka mukana tulee useita helpottavia toimintoja kuten lajittelu-, manipulaatio- sekä hakutoimintoja. Lisäksi luokasta löytyy ToArray() -metodi, joka palauttaa luokan normaalina taulukkona jota Mesh-luokka vaatii. Tämän takia List<T> luokka soveltuukin erinomaisesti geometriadatan sisällyttämiseen (kuva 13, rivit 19-21).

Koska Unityn metodit eivät ole säieturvallisia Unityn pääsäikeestä erotettuna, joutuu kaikki tarvittavat Unity API:sta saatavat tiedot (kuten asetukset PlayerPref:stä sekä positiotiedot transform.position) tallentamaan omaan muuttujaan (kuva 13, rivit 23-28). Chunk-luokka pitää huolen omista säikeistä.

Chunk-luokka lisätään peliobjektiin komponenttina, joten Chunk-luokan on perittävä MonoBehaviour-luokka. Tämä tuo ongelman, jossa emme voi käyttää oletusmuodostinta tai luoda uusia Chunk -luokkia new avainsanalla. Tästä syystä joudutaan ajamaan Chunk-luokan alustusmetodi Init() (kuva 13, rivit 35-57) ennen mitään muuta, jotta tarvittavat muuttujat alustuvat oikein.

```
1   using System.Collections;
2   using System.Collections.Generic;
3   using System.Threading;
4   using Noise;
5   using UnityEngine;
6
7   namespace Map
8   {
9       [RequireComponent(typeof(MeshFilter))]
10      [RequireComponent(typeof(MeshRenderer))]
11      [RequireComponent(typeof(MeshCollider))]
12      public class Chunk : MonoBehaviour
13      {
14          private Mesh _mesh;
15          private MeshFilter _meshFilter;
16          private MeshRenderer _meshRenderer;
17          private MeshCollider _meshCollider;
18
19          private readonly List<Vector3> _verts = new List<Vector3>();
20          private readonly List<int> _tris = new List<int>();
21          private readonly List<Vector2> _uvs = new List<Vector2>();
22
23          // We are unable to access unity stuff outside of unitys main thread so assign needed data to variables
24          private Vector3 _currentPos;
25          private int _chunkSizeX;
26          private int _chunkSizeZ;
27          private int _trackSizeX;
28          private int _meshRatio;
29
30          private Thread _threadMeshData; // Calculates mesh data
31          private bool _calculated;
32
33          private bool _initialized;
34
35          public void Init()
36          {
37              _mesh = new Mesh();
38
39              _meshFilter = GetComponent<MeshFilter>();
40              _meshRenderer = GetComponent<MeshRenderer>();
41              _meshCollider = GetComponent<MeshCollider>();
42
43              ClearMeshData();
44
45              _currentPos = transform.position;
46              _chunkSizeX = Settings.ChunkSizeX;
47              _chunkSizeZ = Settings.ChunkSizeZ;
48              _trackSizeX = Settings.TrackSizeX;
49              _meshRatio = Settings.MeshRatio;
50
51              _meshRenderer.receiveShadows = false;
52
53              _threadMeshData = new Thread(CalculateMeshData);
54              _calculated = false;
55
56              _initialized = true;
57          }
```

Kuva 13. Chunk-luokan muuttujat ja alustusmetodi.

5.4.2 Laskemisen aloittaminen

Koska Chunk-luokka on alustettava oikein ennen sen käyttöä, Chunk-luokka tarkistaa ennen laskemisen aloittamista onko se alustettu (kuva 14, rivit 61-65). Jos ei ole, se antaa virheilmoituksen eikä jatka ajoa.

Jotta voidaan kontrolloida säikeiden määrää, joudumme tekemään tarkistuksen, jossa katsotaan, voidaanko uutta säiettä luoda (kuva 14, rivi 68). Jos ei voida, joudutaan odottamaan. Jotta peli ei jäätyisi odottamisen ajaksi vaan esimerkiksi UI komponentit voisivat vapaasti päivittyä kun Chunk-luokat odottavat lupaa, muutetaan StartCalculation() metodi palauttamaan IEnumerator-objekti (kuva 14, rivi 59). Tämä mahdollistaa kertomaan pelimoottorille, että odotetaan tiettyä asiaa ennen kuin jatketaan kyseisen metodin ajoa. Koska while-silmukassa on “yield return null” komento (kuva 14, rivi 69), while-silmukka tarkistaa joka ruudun päivityksen jälkeen uudelleen voidaanko uutta säiettä luoda.

Kun metodi saa luvan luoda uuden säikeen, se lisää säielaskurille uuden säikeen sekä aloittaa säikeen ajamisen. Metodi sitten odottaa, kunnes säie on valmis ja laskenut geometrian (kuva 14, rivit 76-79). Lopuksi metodi ottaa säielaskurilta kyseisen säikeen pois ja aloittaa ApplyMesh()-metodin, joka tekee geometriadatasta oikean objektin ja käskee pelimoottorin piirtää sen.

```
59      public IEnumerator StartCalculation()
60      {
61          if (!_initialized)
62          {
63              Debug.LogError("Forgot to initialize chunk first");
64              yield break;
65          }
66
67          // Wait until we are allowed to create a new thread
68          while (World.CurrentAmountOfThreads >= Settings.MaxThreads)
69              yield return null;
70
71          World.CurrentAmountOfThreads++;
72
73          _threadMeshData.Start();
74
75          // If we are not yet calculated then check again on next frame
76          while (!_calculated)
77          {
78              yield return null;
79          }
80
81          World.CurrentAmountOfThreads--;
82
83          ApplyMesh();
84      }
```

Kuva 14. Geometrian laskemisen aloittaminen.

5.4.3 Geometrian luominen

Geometria luodaan neljässä vaiheessa:

1. Luodaan maaston vasen puoli kohinaa hyödyntäen ja kopioidaan se erilliseen muuttujaan.
2. Tehdään hiihtoalue. Sen korkeus ei muutu X akselilla ja pisteet ovat samalla tasolla kuin vasemman maaston reuna.
3. Vasemman maaston kopion mukaan tehdään maastosta peilikuva sekä tehdään kolmiot ns. väärään suuntaan, jotta maasto on näkyvillä oikeasta suunnasta.
4. Liitetään maaston peilikopio hiihtoalueen jatkoksi.

Vasen maasto tehdään ilmaan ihmeellisiä temppuja. Tehdään for silmukat X sekä Z koordinaateille. MeshRatio päättää kuinka isot polygonit ovat maastossa. MeshRatiota nostamalla saadaan siis maasto, joka sisältää vähemmän polygoneja mutta on kulmikkaampi.

Ennen verteksien lisäämistä, meidän pitää tietää tulevien verteksien indeksi, jotta voidaan kolmioindeksit luoda oikein. Helpointa on ennen verteksien lisäämistä tallentaa tämän hetkinen verteksien määrä (kuva 15, rivi 93) ja lisätä myöhemmin indeksiin uusien verteksien numero.

Verteksit lisätään _verts-muuttujaan käyttämällä List<T> metodia Add() (kuva 15, rivit 89-112). X sekä Z koordinaatteina käytetään for-silmukoiden X sekä Z arvoa. Korkeusarvo voidaan kysellä aiemmin luodulta BiathlonNoise -luokalta. Korkeusarvon kyselyn koordinaatteihin pitää lisätä Chunk-objektin oikea position maailmassa, jotta jokainen maaston pala ei ole samanlainen (kuva 16). Yhden silmukan iteraation aikana luodaan siis neliö, jonka jokainen kulma on oikeassa paikassa, oikealla korkeudella sekä oikean kokoinen, jonka MeshRatio määrittää. Tästä syystä neliön kulmien koordinaatteihin lisätään MeshRatio. Kun vasen puoli on luotu kokonaan, tallennetaan se erilliseen muuttujaan, josta teemme oikean puolen myöhemmin.

```
86      private void CalculateMeshData()
87      {
88          // Left side of the world
89          for (int x = 0; x < _chunkSizeX / 2; x += _meshRatio)
90          {
91              for (int z = 0; z < _chunkSizeZ; z += _meshRatio)
92              {
93                  int tIndex = _verts.Count;
94
95                  _verts.Add(new Vector3(
96                      x, GetHeight(x, z), z)
97                  );
98                  _verts.Add(new Vector3(
99                      x, GetHeight(x, z + _meshRatio), z + _meshRatio)
100                 );
101                 _verts.Add(new Vector3(
102                     x + _meshRatio, GetHeight(x + _meshRatio, z + _meshRatio), z + _meshRatio)
103                 );
104                 _verts.Add(new Vector3(
105                     x + _meshRatio, GetHeight(x + _meshRatio, z), z)
106                 );
107
108                 CreateTrianglesClockwise(tIndex);
109
110                 CreateDefaultUVs();
111             }
112         }
113
114         Vector3[] worldSide = _verts.ToArray();
```

Kuva 15. Vasemman puoleisen maaston luonti.

```
195     /// Adds real world position to the x/z and gets that points height ...
201     private float GetHeight(float x, float z)
202     {
203         return BiathlonNoise.GetHeight(x + _currentPos.x, z + _currentPos.z);
204     }
```

Kuva 16. GetHeight-metodi, joka lisää maailman position korkeuskyselyihin.

Hiihtoalue tehdään hieman eri tavalla. Koska maaston oikea puoli on vasemman puolen peilikuva, on hiihtoalueen oikean puolen korkeus sama kuin vasemman puolen. Tämä pitää huomioida korkeusarvojen hakemisessa eli jätetään X akselin kasvu huomiotta. Näin saadaan kohina-algoritmilta oikea korkeusarvo, jotta maaston oikea puoli sopii kokonaisuutenaan hiihtoalueen reunalle.

```
116         // Track
117         for (int z = 0; z < _chunkSizeZ; z += _meshRatio)
118         {
119             int tIndex = _verts.Count;
120 
121             _verts.Add(new Vector3(
122                 _chunkSizeX / 2f, GetHeight(_chunkSizeX / 2f, z), z)
123             );
124             _verts.Add(new Vector3(
125                 _chunkSizeX / 2f, GetHeight(_chunkSizeX / 2f, z + _meshRatio), z + _meshRatio)
126             );
127             _verts.Add(new Vector3(
128                 _chunkSizeX / 2 + _trackSizeX, GetHeight(_chunkSizeX / 2f, z + _meshRatio), z + _meshRatio)
129             );
130             _verts.Add(new Vector3(
131                 _chunkSizeX / 2 + _trackSizeX, GetHeight(_chunkSizeX / 2f, z), z)
132             );
133 
134             CreateTrianglesClockwise(tIndex);
135 
136             CreateDefaultUVs();
137         }
```

Kuva 17. Hiihtoalueen luonti.

Jotta oikean puolen maasto sopii hiihtoalueen jatkeeksi, on siitä tehtävä vasemman maaston peilikuva X akselilla. Peilikuvaksi muuttaminen on helpointa muuttamalla jokaisen verteksin X arvo negatiiviseksi (kuva 18, rivit 143-145). Koska tämä siirtää verteksit oikealle, ne täytyy normalisoida takaisin oikeaan paikkaan (kuva 18, rivi 146).

Peilikuvaksi muuttamisen jälkeen voidaan lisätä verteksit meidän dynaamiseen geometriataulukkoon samalla tavalla kuin aiemmin (kuva 18, rivit 149-162). Indeksien määrittämisessä pitää kuitenkin huomioida peilikuva, eli normaalit pitää kääntää ylösalaisin. Jotta vältytään turhalta laskemiselta, voidaan indeksien määrittäminen kääntää vastakellon suuntaan, jolloin Mesh-luokan RecalculateNormals()-metodi laskee normaalit oikein. Jos tätä ei tekisi, olisi oikean puolimmainen maasto näkyvillä vain alhaalta päin katsottuna.

```
142         // Flip the side and move x to the right
143         for (int i = 0; i < worldSide.Length; i++)
144         {
145             worldSide[i] = new Vector3(-worldSide[i].x, worldSide[i].y, worldSide[i].z);
146             worldSide[i] += new Vector3(_chunkSizeX + _trackSizeX, 0f, 0f);
147         }
148 
149         // Add flipped side
150         for(int i = 0; i < worldSide.Length; i += 4)
151         {
152             int tIndex = _verts.Count;
153 
154             _verts.Add(worldSide[i + 0]);
155             _verts.Add(worldSide[i + 1]);
156             _verts.Add(worldSide[i + 2]);
157             _verts.Add(worldSide[i + 3]);
158 
159             // Since X is flipped, we need to inverse the triangle order
160             CreateTrianglesAntiClockwise(tIndex);
161             CreateDefaultUVs();
162         }
163 
164         _calculated = true;
165     }
```

Kuva 18. Oikean puoleisen maaston luonti.

Kuvissa 16, 17 sekä 18 käytetyt CreateTrianglesClockwise-, CreateTrianglesAntiClockwise- sekä CreateDefaultUVs-metodit ovat näkyvillä kuvassa 19. Indeksien määrittäminen tapahtuu tIndex muuttujan mukaan, jossa toinen tekee kolmiot kellon suuntaan ja toinen toiseen suuntaan. CreateDefaultUVs (kuva 19, rivit 187-193) tekee UV koordinaatit niin, että yhteen neliöön tulee yksi tekstuuri kokonaisuudessaan. Mikäli haluttaisiin pitää tekstuurin koko vakiona pituusyksikköihin verrattuna, voidaan koordinaattien 1 arvo korvata MeshRatio-muuttujalla. Näin 2x2 kokoiseen neliöön tulisi yhteensä 4kpl tekstuureja.

```
167     private void CreateTrianglesClockwise(int tIndex)
168     {
169         _tris.Add(tIndex + 0);
170         _tris.Add(tIndex + 1);
171         _tris.Add(tIndex + 2);
172         _tris.Add(tIndex + 2);
173         _tris.Add(tIndex + 3);
174         _tris.Add(tIndex + 0);
175     }
176
177     private void CreateTrianglesAntiClockwise(int tIndex)
178     {
179         _tris.Add(tIndex + 0);
180         _tris.Add(tIndex + 3);
181         _tris.Add(tIndex + 2);
182         _tris.Add(tIndex + 2);
183         _tris.Add(tIndex + 1);
184         _tris.Add(tIndex + 0);
185     }
186
187     private void CreateDefaultUVs()
188     {
189         _uvs.Add(new Vector2(0, 0));
190         _uvs.Add(new Vector2(0, 1));
191         _uvs.Add(new Vector2(1, 1));
192         _uvs.Add(new Vector2(1, 0));
193     }
```

Kuva 19. Indeksien ja UV -koordinaattien luominen.

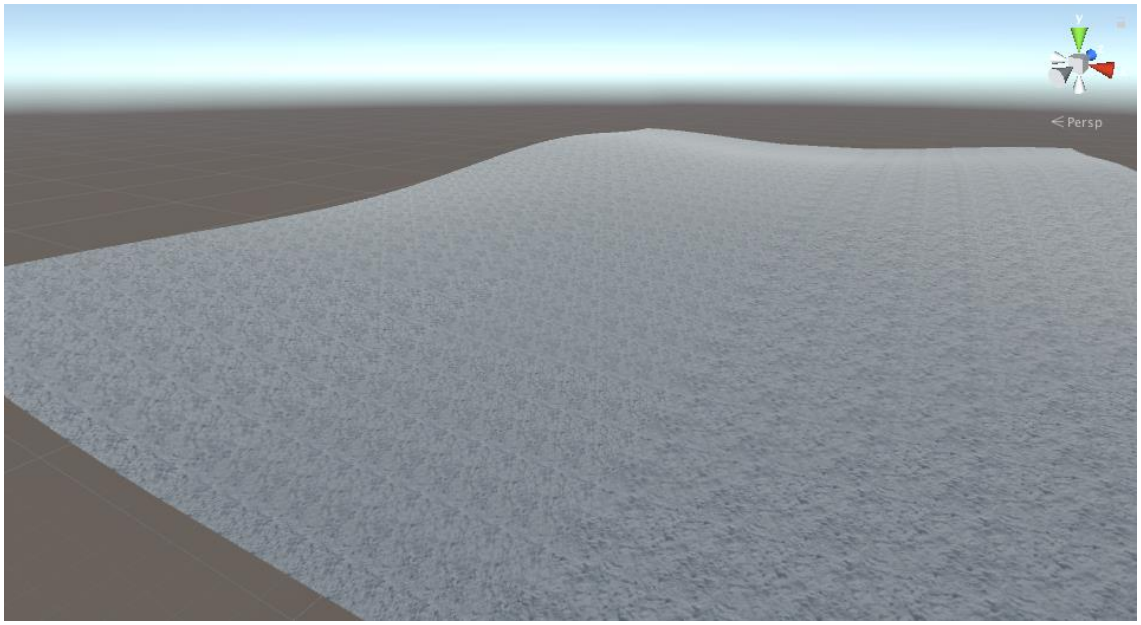
5.4.4 Geometrian piirtäminen peliin

Mesh-luokalle asetetaan tarvittavat verteksit, indeksit sekä UV -koordinaatit. Lisäksi ajetaan Mesh-luokan RecalculateNormals() -metodi, joka laskee automaattisesti normaalit.

Koska geometrian luomisessa huolehdimme kolmioiden määrittämisen aikana että kolmiot ovat oikeaan suuntaan niin RecalculateNormals() -metodi osaa nyt luoda kaikki normaalit oikein. Mesh-objekti asetetaan MeshFilter sekä MeshCollider komponenteille.

```
206         private void ApplyMesh()
207         {
208             _mesh.vertices = _verts.ToArray();
209             _mesh.uv = _uvs.ToArray();
210             _mesh.triangles = _tris.ToArray();
211
212             _mesh.RecalculateNormals();
213
214             _meshFilter.sharedMesh = _mesh;
215             _meshCollider.sharedMesh = _mesh;
216         }
```

Kuva 20. Geometrian piirtäminen peliin.



Kuva 21. Neljän osan hiihtomaailma.

5.5 World-luokka

World luokka alustaa tarvittavat objektit ja parametrit, pitää huolen kentän luomiseen vaativien objektien ajamisesta sekä pitää huolen pelaamisen aikana tarvittavista operaatioista, kuten objektien aktivoinnista/deaktivoinnista. Luokka instantioi ja alustaa kaiken

tarvittavan ennen kuin pelaaja pääsee peliin, joten pelaamisen aikana peli keskittyy ainoastaan pelaamiseen tarvittaviin prosesseihin mahdollistaen mahdollisimman parhaan suorituskyvyn. Samalla voidaan koko hiihtomaailman luomisen aikana näyttää latausruutua tai jopa mahdollisesti mainoksia, joiden alla peli latautuu tuoden illuusion, jossa peli latautuu saumattomasti.

5.5.1 Luokan alustus

Koska säikeiden lukumäärän laskuri on staattinen (kuva 22, rivi 8), nollataan se varmuuden vuoksi joka kerta, kun World luokka aktivoidaan. Muun muassa objektien aktivointi/deaktivointi tarkistaa kentän generaation tilaa tämän muuttujan avulla ja luottaa siihen, että kun muuttujan arvo muuttuu nolla, niin kenttä on generoitu.

Koska Unityn kohtauksien välillä ei voida siirtää dataa suoraan (esim. hiihtomaailmalle ei voida suoraan lähettää kentän numeroa muuttujana), joudutaan turvautumaan staattisiin tai tallennettavaan tietoon, jonka uusi kohtaus hakee. LevelDetail-luokka (kuva 22, rivi 13) on yksinkertaisuudessaan luokka ilman metodeita, joka sisältää tietyn kentän arvot (siemenluku, kentän pituus, esteiden määrä jne.). Nämä arvot ovat muuttumattomia ja ne on asetettu koodissa staattisesti. Kun peli valmistautuu lataamaan seuraavaa hiihtokenttää, se asettaa Settings-luokan avulla seuraan kentän numeron jonka uuden kohtauksen World luokka hakee (kuva 22, rivi 36).

World luokka alustaa myös tarvittavan koon WorldObjects-taulukolle sekä BiathlonNoise-luokan siemenarvon. Lisäksi, jotta oikea määrä kenttiä oikeissa kohdissa aktivoituvat ja muut deaktivoituvat, tarvitaan myös pelaajan position (kuva 22, rivit 37-41).

5.5.2 Chunk-objektien luonti

Jokaisella kentällä on oma hiihtoalueen maksimipituus. Jotta saadaan oikea määrä Chunk-objekteja luotua, on meidän tiedettävä hiihtoalueen pituus sekä Chunk-objektien pituus. Näiden arvojen välillä tehdään for silmukka (kuva 22, rivi 46).

Koska emme suoraan voi käyttää silmukan Z arvoa taulukon indeksiksi joudutaan se muuttamaan oikeaan arvoon. Koska Z arvo saattaa olla 0 ja kyseistä arvoa jaetaan, tehdään ehtolause, joka kitkee nollajaon (kuva 22, rivi 48).

Chunk-luokan tuonti pelimaailmaan onnistuu Unityn Instantiate() -metodilla, jonka avulla voidaan suoraan määrittää myös objektin paikka sekä kierto. Luokka lisätään myös WorldObjects -taulukkoon (kuva 22, rivit 50-52), jotta voimme kyseiselle palalle lisätä myöhemmin objekteja, kuten taloja, esteitä ym. Viimeiseksi alustamme Chunk-luokan sekä aloitamme geometrian laskemisen.

5.5.3 Kentän palojen tarvittava aktivointi/deaktivointi

Jotta peli pyörisi mahdollisimman monella laitteella hyvin, peliin tullaan asettamaan sumu, joka rajoittaa pitkälle näkemisen pelaajalta. Sumun takana olevat objektit voidaan silloin deaktivoida, jolloin laskemisen kuorma pienenee jokaisella ruudunpäivityksellä.

Ehkä helpoin ja yksinkertaisin tapa on vain käydä pala palalta läpi ja verrata palan positio pelaajan positioon sekä katselualueeseen. Mikäli olemme kyseisen alueen ulkopuolella deaktivoimme palan. Jos olemme sen sisäpuolella, aktivoimme palan. (Kuva 22, rivit 60-75).

```
    public static int CurrentAmountOfThreads;

    [SerializeField] private GameObject _chunkFab;

    private LevelDetail _currentLevel;
    private WorldObjects[] _worldObjects;
    private GameObject _player;

    private void Awake()
    {
        EarlyInit();
    }

    private void Start()
    {
        InstantiateChunks();
    }

    private void Update()
    {
        if(CurrentAmountOfThreads == 0)
            HandleObjectActivation();
    }

    private void EarlyInit()
    {
        CurrentAmountOfThreads = 0;

        _currentLevel = Settings.GetCurrentLevel;
        _worldObjects = new WorldObjects[_currentLevel.WorldLength / Settings.ChunkSizeZ];

        BiathlonNoise.Seed = _currentLevel.LevelSeed;

        _player = GameObject.FindGameObjectWithTag("Player");
    }

    private void InstantiateChunks()
    {
        for (int z = 0; z < _currentLevel.WorldLength; z += Settings.ChunkSizeZ)
        {
            int poolIndex = z == 0 ? 0 : z / Settings.ChunkSizeZ;

            // Instantiate chunk and add it to the pool
            Chunk newChunk = Instantiate(_chunkFab, new Vector3(0, 0, z), Quaternion.identity).GetComponent<Chunk>();
            _worldObjects[poolIndex] = new WorldObjects(newChunk.gameObject);

            // Init and start calculation
            newChunk.Init();
            StartCoroutine(newChunk.StartCalculation());
        }
    }

    private void HandleObjectActivation()
    {
        for (int z = 0; z < _currentLevel.WorldLength; z += Settings.ChunkSizeZ)
        {
            int worldObjectIndex = z == 0 ? 0 : z / Settings.ChunkSizeZ;

            if (z >= _player.transform.position.z - Settings.ChunkSizeZ * 2 && z <= _player.transform.position.z + Settings.ViewRange)
            {
                _worldObjects[worldObjectIndex].SetActive(true);
            }
            else
            {
                _worldObjects[worldObjectIndex].SetActive(false);
            }
        }
    }
```

Kuva 22. World-luokka.

6 POHDINTA

Proseduraalisen sisällön generoimiseen ei ole yhtä kultaista tietä vaan ratkaisuja on teoriassa loputtomasti. Mooren lain mukaan laskentateho kaksinkertaistuu noin kahden vuoden välein, jolloin voidaan käyttää aina raskaampia ja monimutkaisempia algoritmeja. Ammatillisilla keskustelupalstoilla tuleekin jatkuvasti uusia algoritmeja, käyttötapoja sekä esimerkkejä vastaan. Vaikka tässä työssä käytiin läpi vain kourallinen eri tapoja ja algoritmeja läpi, ne ovat mielestäni yksinkertaisimmat sekä selkeimmät. Lisäksi niiden valintaan vaikutti se, että kyseiset algoritmit ovat suoraan Unity-pelimoottorissa toteutettu.

Opinnäytetyön tavoitteena oli toteuttaa hiihtomaailman generointijärjestelmän prototyyppi, joka täyttäisi tämän opinnäytetyön alussa esitetyt vaatimukset. Opinnäytetyön puitteissa valmistuikin prototyyppi ja hyvä pohja lopulliselle proseduraaliselle hiihtomaailman generointijärjestelmälle.

Vaikka lopullisessa prototyypissä olikin korjattavaa sekä muutettavaa, se ei kuitenkaan olisi ollut enää mahdollista alkuperäisen aikarajan puitteissa. Vaikka olenkin aiemmin tehnyt vastaavanlaisia prototyyppejä, opein silti uutta mm. moniajosta sekä sain kokemusta yhdistää erinäisiä järjestelmiä ja toimintoja olemassa olevaan tuotteeseen.

Järjestelmää jatkokehitettiin ja lopulta liitettiin Biathlon x1 -peliin, josta tuli lopulta uusi tuote Biathlon x2. Peli on julkaistu sovelluskaupoissa ja on ladattu yli 10 000 kertaa.

LÄHTEET

Goldreich, O. 2006. Computational Complexity: A Conceptual Perspective. Weizmann Institute of Science, Regovot, Israel. Viitattu 20.7.2017. Saatavilla sähköisesti osoitteessa https://users.cs.duke.edu/~reif/courses/complectures/books/G/Gbook.pdf

Microsoft 2014. Minecraft to join Microsoft. Viitattu 20.7.2017. https://news.microsoft.com/2014/09/15/minecraft-to-join-microsoft/

Minecraft. 2017. Sale statistics. Viitattu 20.7.2017. https://minecraft.net/en-us/stats/

Perlin, K. 2001. Improving Noise. New York University. Haettu 20.7.2017 osoitteesta http://mrl.nyu.edu/~perlin/paper445.pdf

Shaker, N.; Togelius, J. & Nelson, M. 2016. Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer. Saatavilla sähköisesti osoitteessa http://pcgbook.com/wp-content/uploads/chapter04.pdf

Unity 2017a. A feature-rich and highly flexible editor. Viitattu 19.7.2017. https://unity3d.com/unity/editor

Unity 2017b. Unity store. Viitattu 19.7.2017. https://store.unity.com.